

Asynchronous Parallel Computing Algorithm implemented in 1D Heat Equation with CUDA

Kooktae Lee^a, Raktim Bhattacharya^a

^a*Laboratory for Uncertainty Quantification*

Department of Aerospace Engineering, Texas A&M University, College Station, TX 77843-3141, USA.

Abstract

In this note, we present the stability as well as performance analysis of asynchronous parallel computing algorithm implemented in 1D heat equation with CUDA. The primary objective of this note lies in dissemination of asynchronous parallel computing algorithm by providing CUDA code for fast and easy implementation. We show that the simulations carried out on nVIDIA GPU device with asynchronous scheme outperforms synchronous parallel computing algorithm. In addition, we also discuss some drawbacks of asynchronous parallel computing algorithms.

Keywords: 1D heat equation, asynchronous parallel computing algorithm, high performance computing, CUDA

1. Introduction

For decades, it has been reported that computing performance in parallel computation can deteriorate due to the synchronization penalty necessarily accompanied by parallel implementation of the given numerical scheme. Thus, there is a trend to relax this synchronization latency by adopting alternative approaches and techniques such as relaxed synchronization [1, 2] or *asynchronous parallel computing algorithm* [3, 4, 5, 6, 7, 8]. Although the asynchronous parallel computing algorithm has arisen to overcome the synchronization bottleneck, and hence speed up the computation, the randomness of asynchrony incurs unpredictability of the solution, which in turn leads to numerical inaccuracy of the solution or even instability in the worst case. Therefore, asynchronous algorithms have to be analyzed rigorously before it is fully implemented.

In [7], we have developed mathematical proofs for stability, rate of convergence, and error probability of asynchronous 1D heat equation via dynamical system framework (especially, the switched system framework [9, 10]). All the results in this note are based on our previous research works [7]. Thus, this note aims at testing asynchronous scheme in 1D heat equation with CUDA rather than developing theory and proof. In particular, we mainly focus on easy implementation of asynchronous algorithm by providing the CUDA code, to achieve high performance computing. In summary, the primary goal of this note

Email addresses: animodor@tamu.edu (Kooktae Lee), raktim@tamu.edu (Raktim Bhattacharya)

lies in dissemination of the asynchronous parallel computing algorithm to enhance the computing performance of conventional parallel computation. For more details regarding the theoretical development, the readers may refer to [7]. The simulations carried out on nVIDIA GPU device with CUDA present the stability result and performance analysis as well. In the last section, we also discuss some drawbacks of asynchronous parallel computing algorithm.

2. Problem Formulation

Consider 1D heat equation, of which partial differential equation (PDE) is given by

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}, \quad t \geq 0, \quad (1)$$

where u is the time and space-varying state of the temperature, and t and x are continuous time and space respectively. The constant $\alpha > 0$ is the thermal diffusivity of the given material.

The PDE is solved numerically using the finite difference method by Euler explicit scheme, with a forward difference in time and a central difference in space. Adopting this finite difference method leads to

$$\begin{aligned} \frac{\partial u}{\partial t} &\approx \frac{u_i(k+1) - u_i(k)}{\Delta t}, \\ \frac{\partial^2 u}{\partial x^2} &\approx \frac{u_{i+1}(k) - 2u_i(k) + u_{i-1}(k)}{\Delta x^2}, \end{aligned}$$

where $k \in \{0, 1, 2, \dots\}$ is the discrete-time index and u_i , $i = 1, 2, \dots, N$, is the temperature value at i^{th} grid space point with total N numbers of the grid point. Thus (1) is approximated as

$$\frac{u_i(k+1) - u_i(k)}{\Delta t} = \alpha \left(\frac{u_{i+1}(k) - 2u_i(k) + u_{i-1}(k)}{\Delta x^2} \right), \quad (2)$$

where the symbols Δt and Δx denote the sampling time and the grid resolution in space, respectively. Further, if we define a constant $r \triangleq \alpha \frac{\Delta t}{\Delta x^2}$, then (2) can be written as

$$u_i(k+1) = ru_{i+1}(k) + (1 - 2r)u_i(k) + ru_{i-1}(k), \quad (3)$$

where the parameter r lies in between 0 and 0.5 for the numerical stability (see pp. 18, [11]). Also, we consider the Dirichlet boundary condition (see pp. 150, [12]), i.e., the temperature at each end-point is invariant in time as follows:

$$u_1(k) = c_1, \quad u_N(k) = c_2, \quad \forall k$$

with some constants c_1 and c_2 .

Fig. 1 illustrates the numerical scheme over the discretized 1D spatial domain. A typical *synchronous* parallel implementation of this numerical scheme assigns several of these grid points to each processing element (PE). The updates for the temperature at the grid points assigned to each PE, occur in parallel. However, at every time step k , the

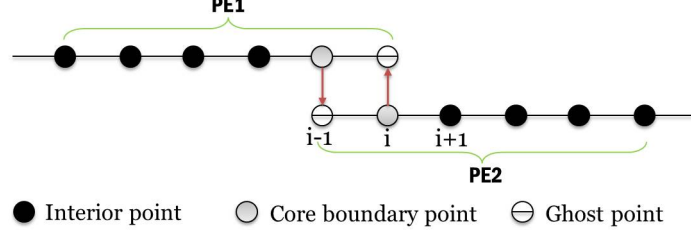


Figure 1: Discretized one-dimensional domain with an asynchronous numerical algorithm. The PE denotes a group of grid points, assigned to each core.

data associated with the boundary grid points, where the communication is necessary are synchronized, and used to compute $u_i(k+1)$. This synchronization across PEs is slow, especially for massively parallel systems (estimates of idle time due to this synchronization give figures of up to 80% of the total time taken for the simulation as idle time).

3. Asynchronous Parallel Computing Algorithm

Recently, an alternative implementation – *asynchronous* algorithm – has been proposed. In this implementation, the updates in a PE occur without waiting for the other PEs to finish and their results to be synchronized. The data update across PEs occurs sporadically and independently. This asynchrony directly affects the update equation for the boundary points, as they depend on the grid points across PEs. For these points, the update is performed with the most recent available value, typically stored in a buffer. The effect of this asynchrony then propagates to other grid points. Within a PE, we assume there is no asynchrony and data is available in a common memory.

Thus, the asynchronous numerical scheme corresponding to (3) is given by

$$u_i(k+1) = ru_{i+1}(k_{i+1}^*) + (1-2r)u_i(k) + ru_{i-1}(k_{i-1}^*), \quad (4)$$

where $k_i^* \in \{k, k-1, k-2, \dots, k-q+1\}$, $i = 1, 2, \dots, N$, denotes the randomness caused by communication delays between PEs. The subscript i in k_i^* depicts that each grid space point may have different time delays. The parameter q is the length of a buffer that every core maintains to store data transmitted from the other cores.

In the following section, we provide the CUDA codes for both synchronous and asynchronous implementation of 1D heat equation.

4. CUDA Code

At first, we take a look at the synchronous code. In the parallel implementation of (3), only time-loop for index k is necessary, since the space-loop for index i can be carried out in parallel. To enforce synchronization, the parallel computation in space index i is performed only once in CUDA kernel, and then we repeat this process thereafter in the *main* through discrete-time iteration. After executing *kernel*, it is guaranteed that each

$u[i]$ value is computed and copied to the host memory. Thus, the synchronization is imposed at each instance. The time-loop is then given in the *main*, instead of *kernel*, for the synchronization issue. This would be a naive way to synchronize and alternative techniques can be also applied for synchronization such as ‘`__syncthreads()`’, which may further increase computing performance.

- **Synchronous Algorithm**

```
--global-- void kernel(float* u){
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    if( i > 0 && i < N-1){
        u[i] = r*(u[i+1]-2*u[i]+a[i-1]) + u[i];
    }
}

int main(){
    float *u, *uDev;
    int size1 = N*sizeof(float);

    cudaMalloc((void**) &uDev, size1);
    u = (float*)malloc(size1);

    // initial condition
    for(int i=0; i<N; i++)
    {
        u[i] = pow(cos( 3*PI/2*i/(N-1) ),2); // cosine func.
    }

    int dimThreads = Npts;
    int dimBlock = Npts/dimThreads;
    for(int k=0; k<kend; k++) // time-loop
    {
        cudaMemcpy(uDev, u, size1, cudaMemcpyHostToDevice);
        kernel<<<dimBlock, dimThreads>>>(uDev);
        cudaMemcpy(u, uDev, size1, cudaMemcpyDeviceToHost);
    }
    free(u);
    cudaFree(uDev);
    return 0;
}
```

Next, we consider the asynchronous code. The major difference between synchronous and asynchronous codes are the placement of the time-loop. In this asynchronous code, the time-loop is imposed in the *kernel*, and hence each $u[i]$ can be updated asynchronously without any *barrier* for synchronization. The purpose of new variables ‘ v ’ and ‘ $vDev$ ’ in the asynchronous code is to keep track of $u[i]$ in time, since one cannot save the history of $u[i]$ while processing *kernel*.

- **Asynchronous Algorithm**

```
--global-- void kernel(float* u, float* v){

    int i = blockIdx.x*blockDim.x + threadIdx.x;
```

```

    for(int k=0;k<kend;k++) // time-loop
    {
        if( i > 0 && i < N-1){
            u[i] = r*(u[i+1]-2*u[i]+u[i-1]) + u[i];
        }
        v[N*k+i] = u[i];
    }
}

int main(){
    float *u, *uDev, *v, *vDev;
    int size1 = N*sizeof(float);
    int size2 = N*kend*sizeof(float);

    cudaMalloc((void**) &uDev, size1);
    cudaMalloc((void**) &vDev, size2);
    u = (float*)malloc(size1);
    v = (float*)malloc(size2);

    // initial condition
    for(int i=0;i<N;i++)
    {
        u[i] = pow(cos( 3*PI/2*i/(N-1) ),2); // cosine func.
    }

    cudaMemcpy(uDev,u,size1,cudaMemcpyHostToDevice);
    int dimThreads = Npts;
    int dimBlock = Npts/dimThreads;
    kernel<<<dimBlock,dimThreads>>>(uDev, vDev);
    cudaMemcpy(v,vDev,size2,cudaMemcpyDeviceToHost);

    free(u);
    free(v);
    cudaFree(uDev);
    cudaFree(vDev);
    return 0;
}

```

5. Simulation Results

We implemented the asynchronous parallel algorithm with **CUDA C++ programming** on **nVIDIA Tesla™ C2050 GPU**, which has **448 CUDA cores**. The simulations were performed with the following parameters:

- Simulation Parameters:

$$\Delta x = 0.1, \Delta t = 0.01, \alpha = 0.5, r = \alpha \frac{\Delta t}{\Delta x^2} = 0.5$$

$$I.C.: \quad u_i = \cos^2\left(\frac{3\pi i}{2(N-1)}\right), \quad i = 1, 2, \dots, N$$

$$B.C.: \quad u_1(k) = 1, \quad u_N(k) = 0, \quad \forall k$$

- Number of PEs: $N = 100$.

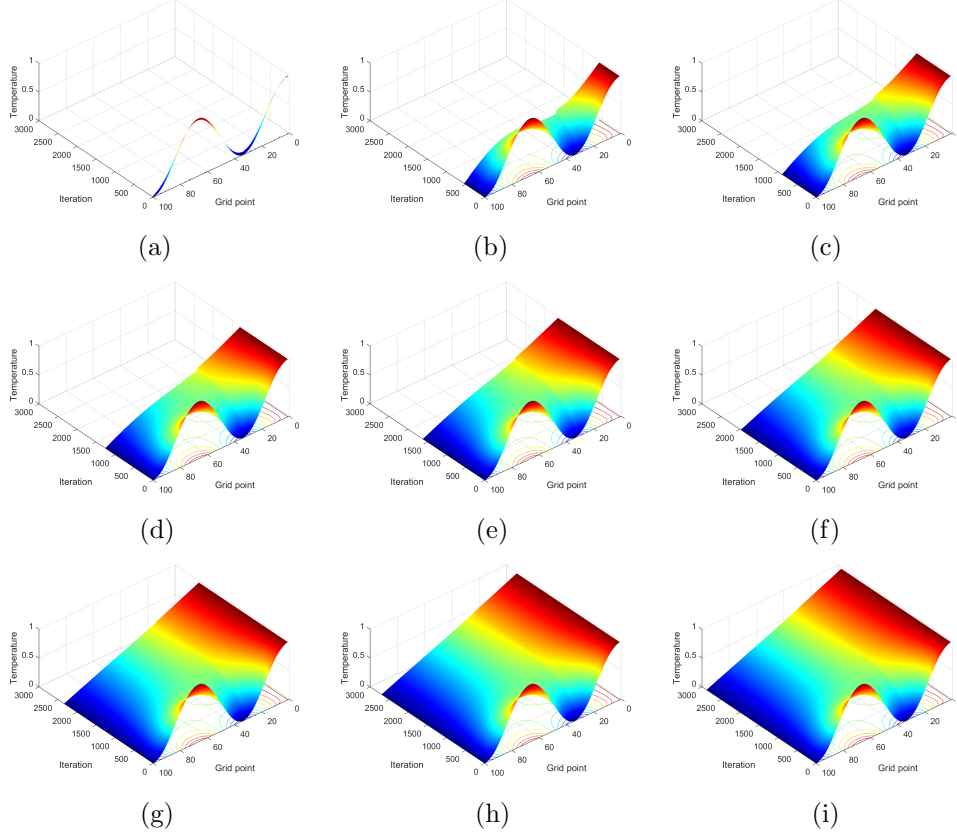


Figure 2: The spatio-temporal evolution of 1D heat equation using asynchronous parallel iterative method

- Number of grid points in PE: $n = 1$

In this simulation, we consider that each grid point is assigned to each PE (CUDA core in here) as defined $n = 1$ above. Thus, each CUDA core updates the value of u_i .

5.1. Spatio-temporal evolution of temperature

For a given initial temperature, the spatio-temporal evolution of the state is presented in Fig. 2. As time k increases, the curved shape of the temperature, given as a cosine square function initially, flattens out.

In Fig. 3, the ensemble of the trajectories is shown for the asynchronous algorithm. The solid lines show the trajectories of total 300 simulations. Due to the randomness in the asynchronous algorithm, the trajectories differ from each other. As we already proved in [7], the asynchronous scheme for 1D heat equation is numerically stable, which is also verified in this simulation with CUDA.

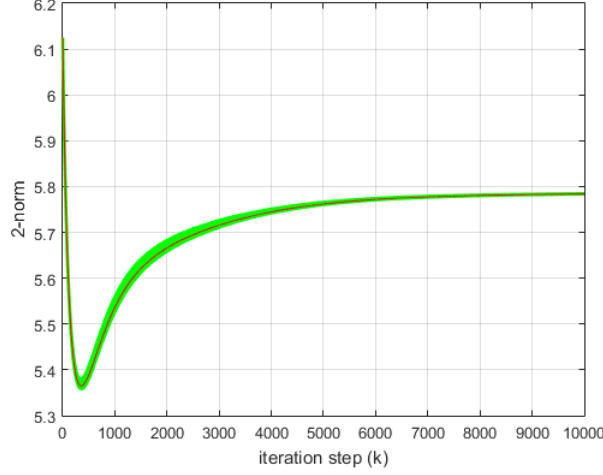


Figure 3: The ensemble (green lines) of 300 trajectories in 2-norm values with regards to the discrete temperature $u(k) = [u_1^T(k), \dots, u_N^T(k)]^T$ for asynchronous scheme and their corresponding mean value (red line).

5.2. Performance analysis

For comparative purposes, we carried out multiple simulations while increasing the number of grid points in 1D heat equation. Fig. 4 and 5 present the total execution time, which describes how much physical time has elapsed up to the given iteration steps. For both synchronous and asynchronous cases, the total execution time increases as the number of grid points N increases. This is because the computation cost grows with proportional to the problem size. As evidently shown in Fig. 4 and 5, the asynchronous algorithm drastically speeds up the execution time, and hence benefits the computing performance. In almost all cases, it is observed that asynchronous scheme outperforms synchronous scheme. Particularly when $N = 100$, asynchronous scheme brought more than $30\times$ speedup compared to the synchronous scheme, which is substantial and outstanding acceleration.

5.3. Intriguing Remarks

The stability result for 1D heat equation with Dirichlet boundary condition is given in [7]. In [7], we guaranteed that starting from given initial condition, temperature converges to unique steady-state distribution after sufficiently large iterations, *regardless of* asynchrony. This implies that the steady-state solution obtained by asynchronous scheme is independent of asynchronous behavior even though it affects on the transient jitters. Note that the notion of stability in this case means that the temperature distribution does not diverge as well as converges to *unique* steady-state value.

However, 1D heat equation with *periodic* boundary condition implemented in asynchronous scheme does not have unique steady-state solution. Here the periodic boundary condition denotes the case in which temperature at both end-points depends on each

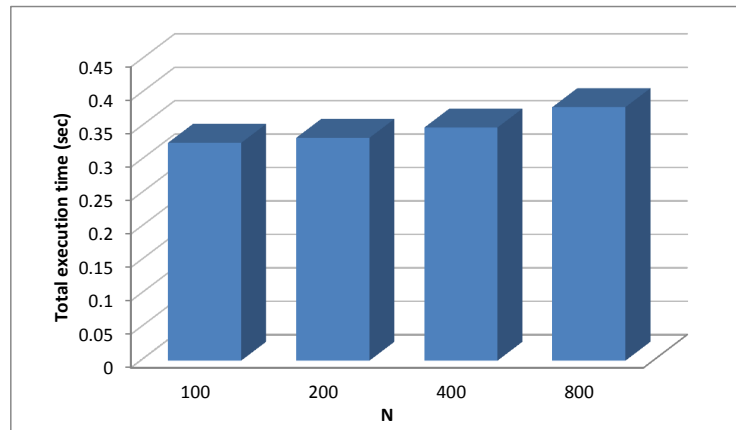


Figure 4: Total execution time for *synchronous* parallel computing algorithm with respect to variation in the number of grid points N

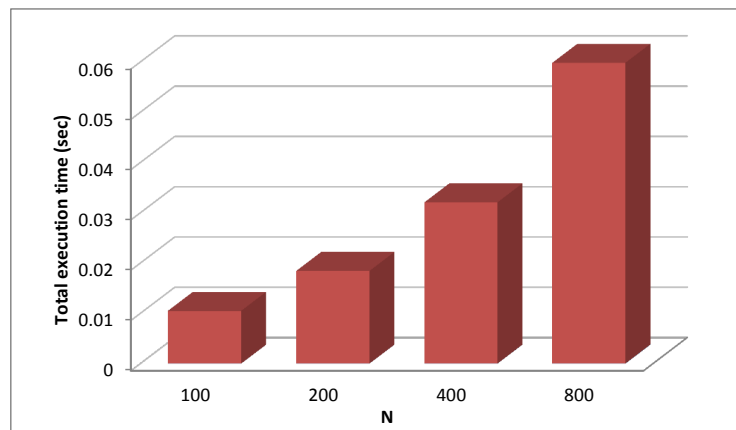


Figure 5: Total execution time for *asynchronous* parallel computing algorithm with respect to variation in the number of grid points N

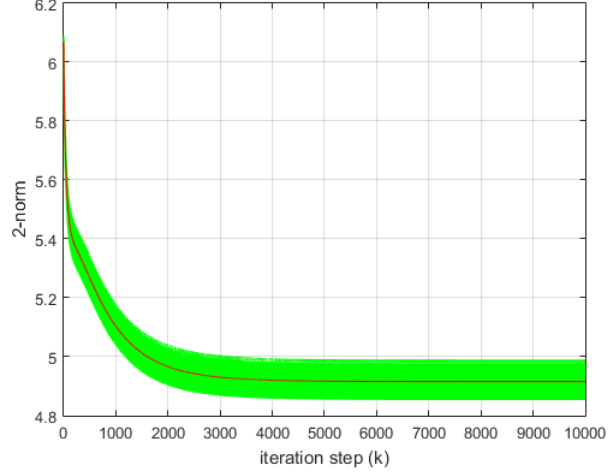


Figure 6: The ensemble (green lines) of 300 trajectories in 2-norm values and their corresponding mean value (red line) for *periodic* 1D heat equation with implementation of asynchronous parallel computing algorithm.

other, i.e.,

$$\begin{aligned} u_1(k) &= r(u_2(k) - 2u_1(k) + u_N(k)) + u_1(k), \\ u_N(k) &= r(u_1(k) - 2u_N(k) + u_{N-1}(k)) + u_N(k). \end{aligned}$$

The CUDA code below presents the asynchronous 1D heat equation with periodic boundary condition.

```
--global-- void kernel(float* u, float* v){
    int i = blockIdx.x*blockDim.x + threadIdx.x;

    for(int k=0;k<kend;k++){
        if( i > 0 && i < N-1){
            u[i] = r*(u[i+1]-2*u[i]+u[i-1]) + u[i];
        }
        /* periodic boundary condition */
        u[0] = r*(u[1]-2*u[0]+u[N-1]) + u[0];
        u[N-1] = r*(u[0]-2*u[N-1]+u[N-2]) + u[N-1];

        v[N*k+i] = u[i];
    }
}
```

In Fig. 6, we demonstrate multiple simulations starting from same initial condition given by cosine function as in the previous case. The ensemble of 300 trajectories spreads out and does not converge to unique value, which is different from asynchronous 1D heat equation with Dirichlet boundary condition. Thus, in the case of periodic 1D heat equation, it is observed that asynchronous parallel computing algorithm is numerically stable (in the sense that the solution does not diverge), but the solution is not unique.

It is very interesting to see that even with exactly same PDE, same finite difference scheme, and same initial condition, one may expect different convergence results for different boundary conditions. Note that the synchronous solution of periodic 1D heat equation always ends up with unique steady-state temperature distribution, since there is no randomness in synchronous scheme. This implies that incorrect information would be delivered by asynchronous scheme. In fact, the convergence or stability of asynchronous scheme is problem-dependent! (i.e., case by case). Therefore, asynchronous parallel computing algorithms give rise to a tradeoff issue between speedup and accuracy and hence, implementation of asynchronous parallel computing algorithms requires rigorous mathematical analysis before it is fully implemented.

References

- [1] J.-S. Kim, S. Ha, C. S. Jhon, Relaxed barrier synchronization for the bsp model of computation on message-passing architectures, *Information Processing Letters* 66 (5) (1998) 247–253.
- [2] L. Renganarayana, V. Srinivasan, R. Nair, D. Prener, Programming with relaxed synchronization, in: *Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability*, ACM, 2012, pp. 41–50.
- [3] D. P. Bertsekas, J. N. Tsitsiklis, *Parallel and distributed computation: numerical methods*, Prentice-Hall, Inc., 1989.
- [4] A. Frommer, D. B. Szyld, On asynchronous iterations, *Journal of computational and applied mathematics* 123 (1) (2000) 201–216.
- [5] J. M. Bahi, S. Contassot-Vivier, R. Couturier, F. Vernier, A decentralized convergence detection algorithm for asynchronous parallel iterative algorithms, *Parallel and Distributed Systems, IEEE Transactions on* 16 (1) (2005) 4–13.
- [6] G. C. Fox, R. D. Williams, G. C. Messina, *Parallel Computing Works!*, Morgan Kaufmann, 2014.
- [7] K. Lee, R. Bhattacharya, V. Gupta, A switched dynamical system framework for analysis of massively parallel asynchronous numerical algorithm, in: *American Control Conference (ACC)*, 2015. *Proceedings of the 2015, IEEE*, 2015, pp. 1095–1100.
- [8] K. Lee, R. Bhattacharya, On the convergence analysis of asynchronous distributed quadratic programming via dual decomposition, *arXiv preprint arXiv:1506.05485*.
- [9] K. Lee, R. Bhattacharya, Stability analysis of large-scale distributed networked control systems with random communication delays: A switched system approach, *Systems & Control Letters* 85 (2015) 77–83.
- [10] K. Lee, A. Halder, R. Bhattacharya, Performance and robustness analysis of stochastic jump linear systems using wasserstein metric, *Automatica* 51 (2015) 341–347.
- [11] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*, Oxford university press, 1985.
- [12] R. H. Pletcher, J. C. Tannehill, D. Anderson, *Computational fluid mechanics and heat transfer*, CRC Press, 2012.